24 | 压测平台：如何改造分布式调度平台？

2021-12-11 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼
时长 21:24 大小 19.61M

你好，我是高楼。

这节课，我们继续来讲分布式压测平台的改造。在上一讲，我们已经知道了对象存储和性能监控的实现步骤，这节课，我们来看看分布式调度的改造。

分布式调度平台的选型

在 Java 技术栈中，传统的任务调度实现方案，比如 Quartz、Timer，都存在一些问题，例如：

仅单点，不支持集群任务分发；

无法统计任务执行数据；

没有监控任务执行状态；

没有任务异常告警；

没有 web 后台管理。

但在如今的全链路压测场景中，大部分的压测平台都是分布式架构，它对任务调度的需求也更高，例如：

当同一个压力机或多个压力机的下发任务互斥时，需要做到统一的调度；

下发的任务需要支持高可用、实时监控、异常告警等功能；

压力机集群需要支持动态、协同进行压测，能应付高并发海量流量的场景；

需要统一的管理并能够追踪各压力机任务执行结果，需要快速记录各种任务的属性数据。

也就是说，传统的任务调度实现方案已经不能满足全链路压测平台的需求了。这时候，我们就需要一个分布式任务调度平台。目前业界比较主流的中间件有哪些呢？

	Elastic-Job	XXL-JOB	Quartz
定时调度	Cron	Cron	Cron
依赖组件	JDK、Zookeeper	MySQL、JDK	MySQL、JDK
高可用	通过 Zookeeper 的注册与发现，可以动态地添加服务器。	基于竞争数据库锁，保证只有一个节点执行任务，支持水平扩容。可以手动增加定时任务，启动和暂停任务，有监控。	多节点部署，通过竞争数据库锁来保证只有一个节点执行任务。
任务分片	✔	✔	✗
管理界面	✔	✔	✗
任务治理	• 执行记录：无 • 运行大盘：有 • 运行日志：有	• 执行记录：有 • 运行大盘：有 • 运行日志：有	✗
任务类型	• Java • Shell	• Java • Shell • Python • PHP • Node.js	• Java
使用成本	• DB • 至少 3 个 ZooKeeper • 2 个 Console • 多个 Server • 人工运维成本	• DB • 1 个调度中心 • 多个执行器 • 人工运维成本	• DB • 多个 Server • 人工运维成本
难易程度	简单	简单	略复杂
异常告警	需自研	邮件	无
高级功能	弹性扩容，多种作业模式，失效转移，运行状态收集，多线程处理数据，幂等性，容错处理，Spring 命名空间支持	弹性扩容，分片广播，故障转移，Rolling 实时日志，GLUE（支持在线编辑代码，免发布），任务进度监控，任务依赖，数据加密，邮件报警，运行报表，国际化	无
公司	当当	个人	OpenSymphony
项目活跃度	不太活跃	较活跃	活跃

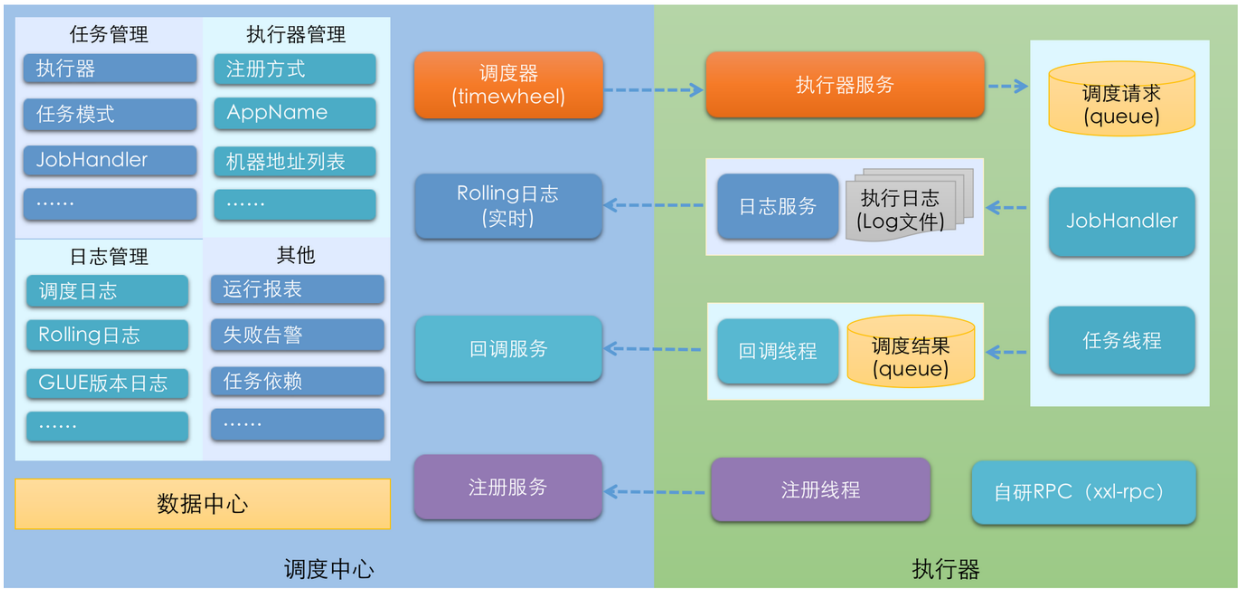
极客时间

目前业界比较主流的中间件主要有Elastic-Job、XXL-JOB和 Quartz 三种。由于 XXL-JOB 的热度足够（目前关于 XXL-JOB 的 GitHub Stars 有 20.1k 个，登记在用的公司有 446 个），另外它的开发迅速、学习简单、轻量级、易扩展，这些特性都挺适合我们这个全链路压测项目的。综合考虑后，最后我们选择了 **XXL-JOB**。

XXL-JOB 基本概念

既然选定了 XXL-JOB，就要先了解一下 XXL-JOB 的基本结构。

下面是一张 XXL-JOB （V2.1.0）的官网架构图。



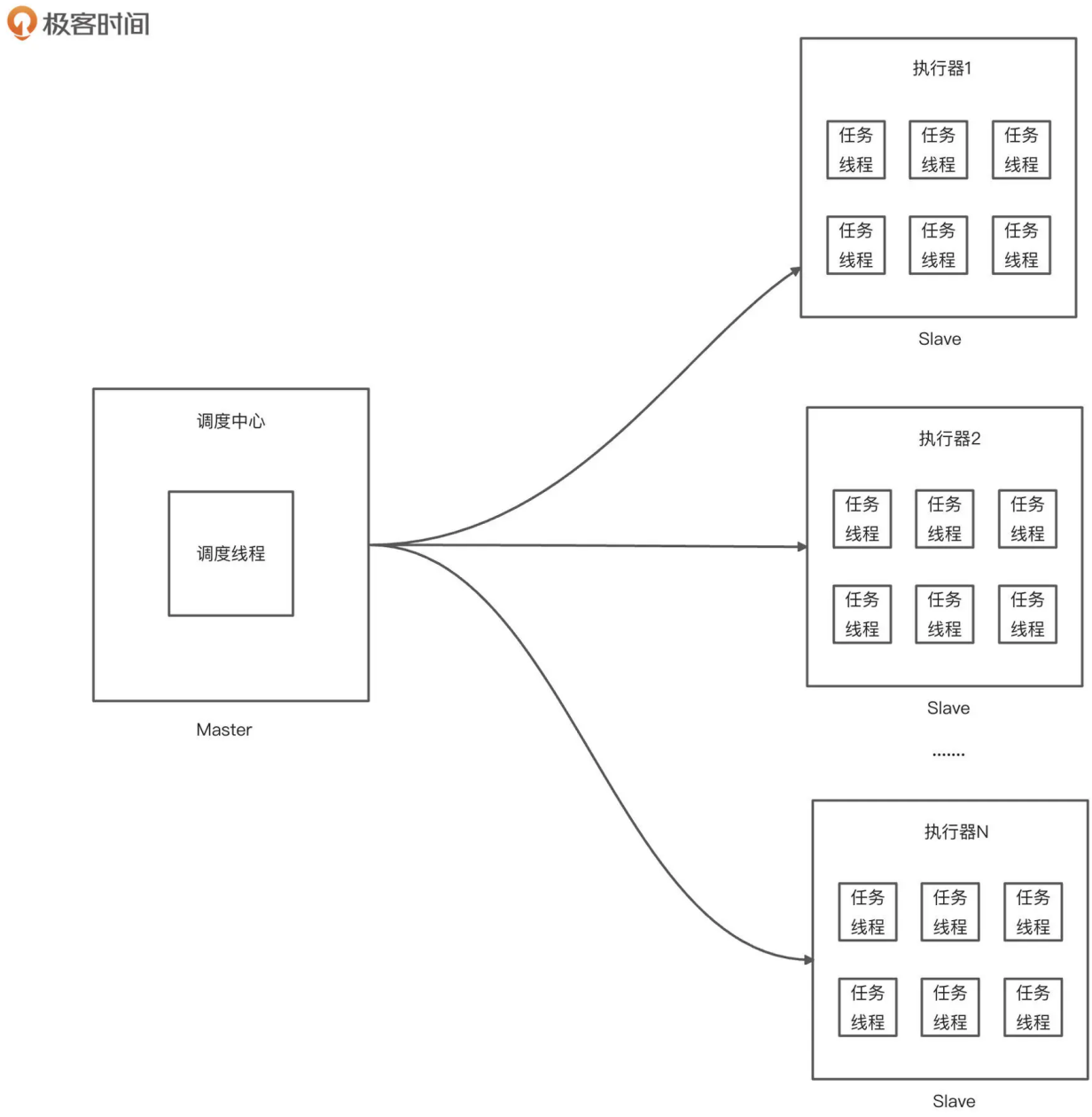
XXL-JOB架构图 v2.1.0

从图中我们也可以很直观地看到，XXL-JOB 主要包括两大模块：

调度模块（调度中心）：负责管理调度信息，按照调度配置发出调度请求，它自身不承担业务代码。这种调度系统与任务解耦的方式，提高了系统可用性和稳定性，也让调度系统性能不再受限于任务模块。另外，调度中心支持可视化、简单且动态地管理调度信息，例如任务新建、更新、删除、GLUE 开发和任务报警等，所有上述操作都会实时生效。调度中心还支持监控调度结果以及执行日志，支持执行器 Failover。

执行模块（执行器）：负责接收调度请求并执行任务逻辑。它可以让任务模块专注于任务的执行等操作，让开发和维护更加简单高效。

为了方便理解，我这里画了一个更加精简的逻辑图：



可以看到，调度器和执行器是独立的。调度器决定任务的调度，并且通过 HTTP 的方式调用执行器接口，执行相应任务。

分布式调度平台的落地思路

那么，我们要怎么通过分布式调度平台来满足我们的需求呢？要清楚这一点，我们就要先理清执行器在流量平台的主要用途，它可以：

1. 管理 GoReplay 的生命周期 ，可以调起和终止 GoReplay 程序；
2. 回传状态（在 GoReplay 启动、结束、或其他标志性事件结束后都会向调度服务回传状态）；

3. 对录制和回放的相关文件进行下载、处理和回传；
4. 实时 Rolling 日志，记录 GoReplay 输出的状态数据，便于实时监控。

对于我们分布式流量平台来说，主要的需求就是流量录制和流量回放操作的调度。我们先来熟悉一下这两种调度方式的落地思路，再来进行具体的实战演示。

流量录制主要过程：

1. 用户设置录制参数，定义录制命令，创建录制任务；
2. 调度中心生成录制任务，并下发到对应网关的执行器上；
3. 执行器收到录制命令，回传状态给调度中心，随即执行对应录制命令，拉起 GoReplay 程序；
4. 录制结束后，GoReplay 程序退出，执行器回传状态给调度中心；
5. 调度中心判定录制任务结束后，下发上传文件子任务到相关执行器上，执行器执行上传命令将对应的流量文件通过 HTTP 上传到对象存储中；
6. 流量录制任务结束。

流量回放的主要过程：

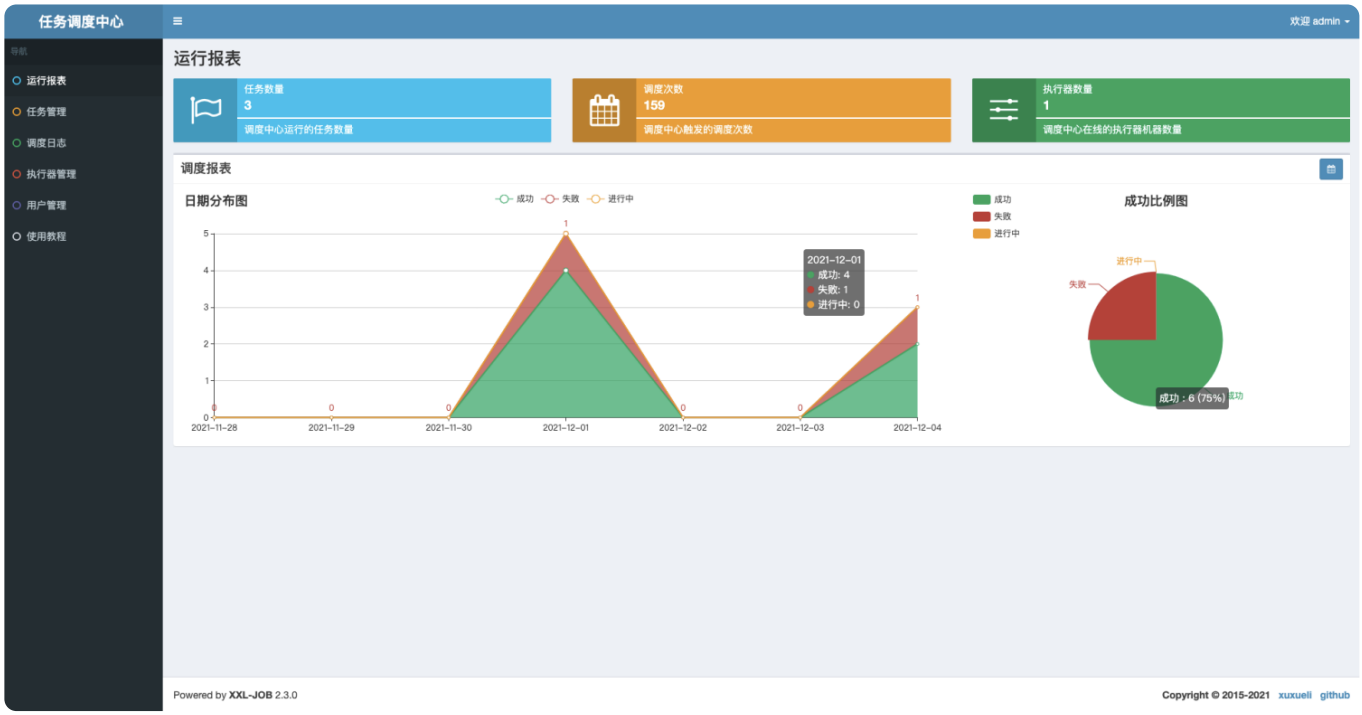
1. 用户设置回放参数，定义回放命令，创建回放任务；
2. 调度中心生成回放任务，并下发到对应压力机的执行器上；
3. 执行器收到回放命令，回传状态给调度服务，随即拉起 GoReplay 程序；
4. 回放结束，GoReplay 退出，执行器回传状态给调度中心；
5. 流量回放任务结束。

好了，了解了流量平台的调度使用过程，我们就来一起看看具体怎样去落地。

分布式调度平台的落地实践

首先，我们需要搭建 XXL-JOB 调度平台，具体的搭建和系统初始化步骤你可以参考官方文档。

搭建成功后，登录调度中心，运行界面如下：



调度中心支持集群部署，这可以提升调度系统的容灾和可用性。不过，在调度中心集群部署时我有几点建议：

DB 配置保持一致；

集群机器时钟保持一致（单机集群忽视）；

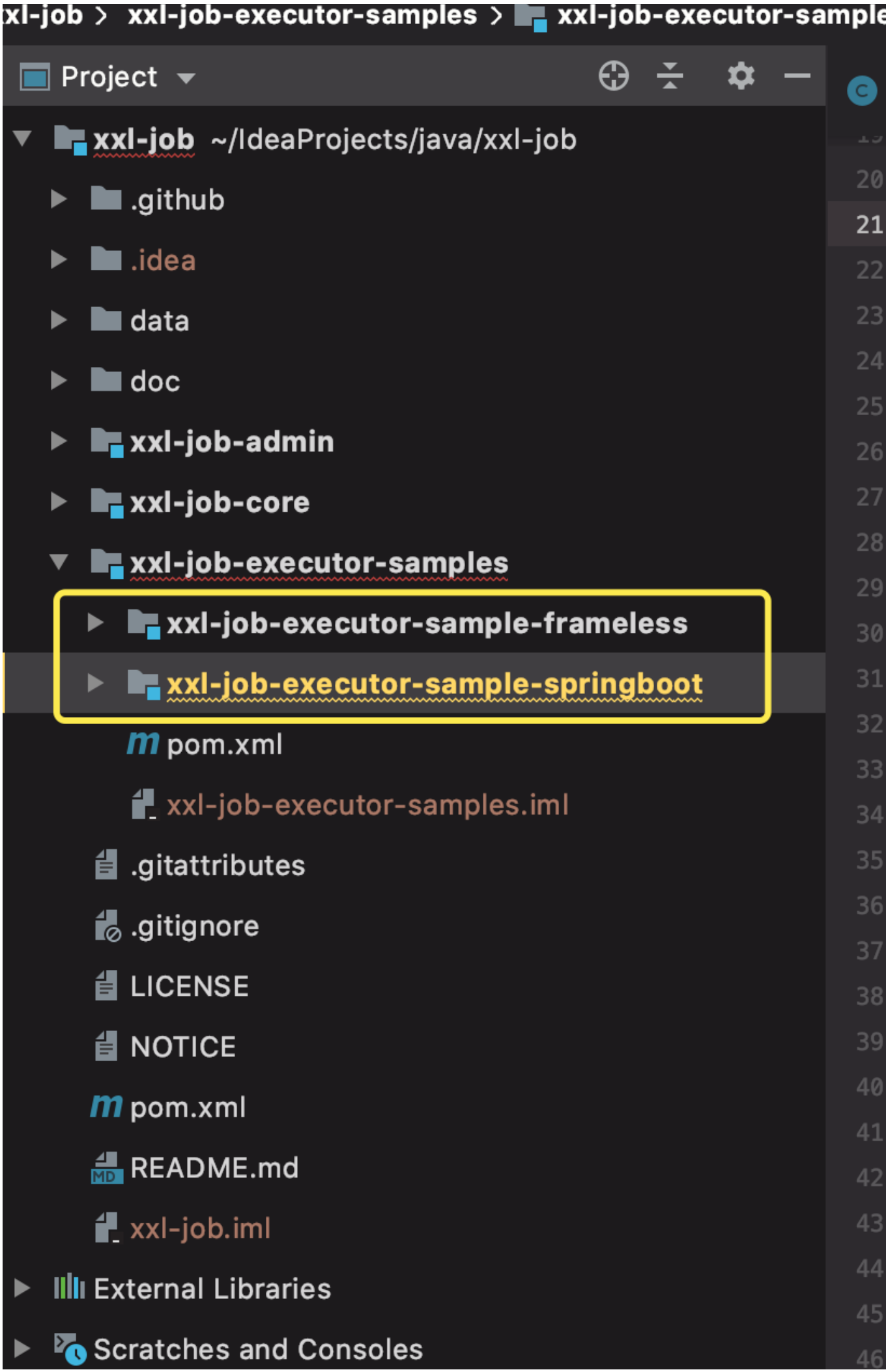
通过 Nginx 为调度中心集群做负载均衡，分配域名（调度中心访问、执行器回调配置、调用 API 服务等操作均通过该域名进行）。

有了调度平台，我们就可以进行具体的实践了：

创建执行器

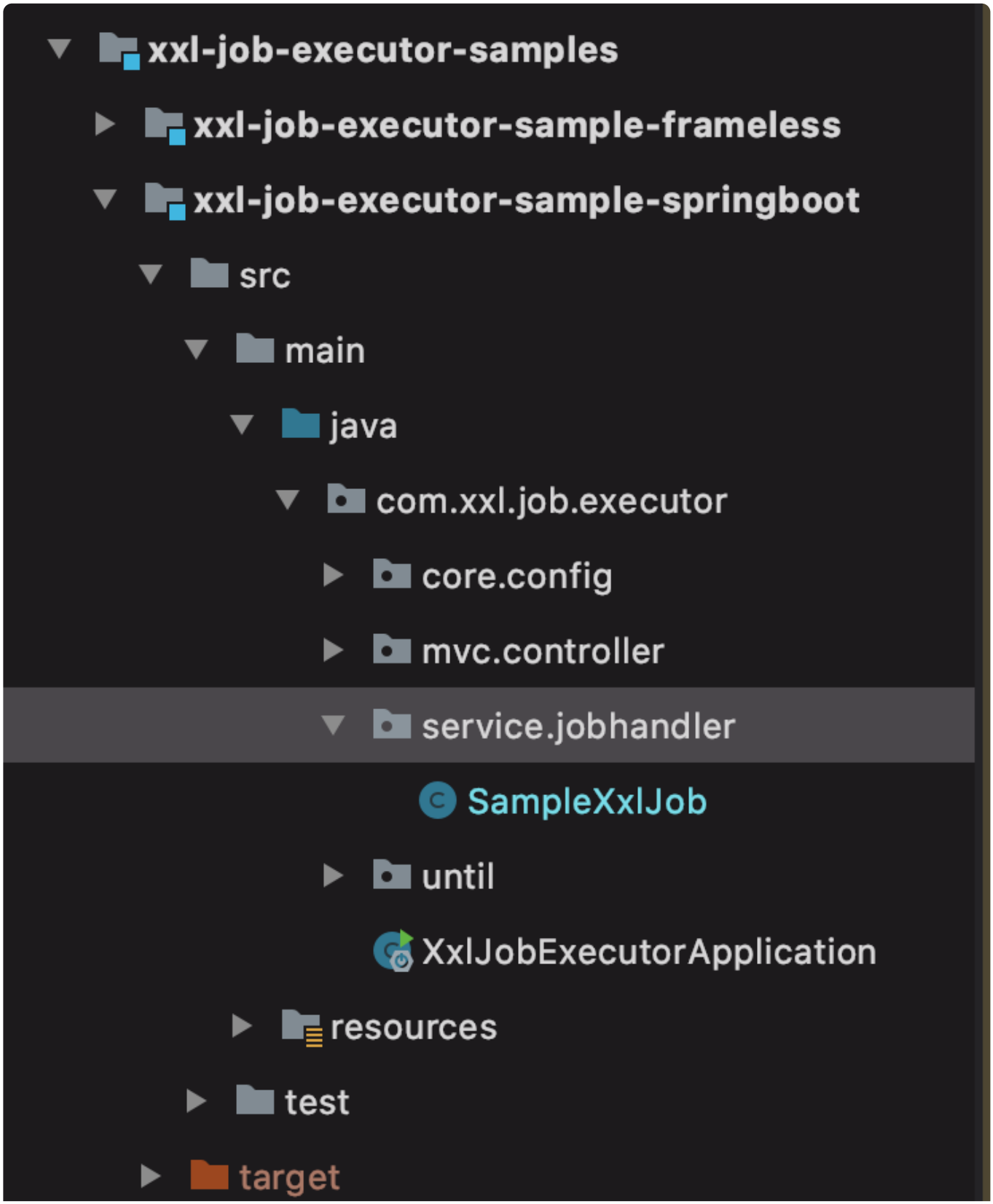
执行器相当于是包工头，它可以是一个单独项目，也可以集成在其它业务项目里面。要注意的是，执行器首先需要注册到调度中心，这样调度中心才能发现执行器。执行器的执行结果，也需要通过回调的方式反馈给调度中心。

在源码的 xxl-job-executor-samples 里，有 2 个执行的 demo 教我们如何去创建执行器。


xxl-job > xxl-job-executor-samples > xxl-job-executor-sample
Project
xxl-job ~/IdeaProjects/java/xxl-job
.github
.idea
data
doc
xxl-job-admin
xxl-job-core
xxl-job-executor-samples
xxl-job-executor-sample-frameless
xxl-job-executor-sample-springboot
pom.xml
xxl-job-executor-samples.iml
.gitattributes
.gitignore
LICENSE
NOTICE
pom.xml
README.md
xxl-job.iml
External Libraries
Scratches and Consoles

```
47
48
```

这里我用 SpringBoot 项目用来举个例子。首先，要单独拷一个项目出来，如果你想在自己的项目里集成的话，同样也是参考这个 demo，只要在项目 pom 里加上 xxl-job-core 的依赖即可。

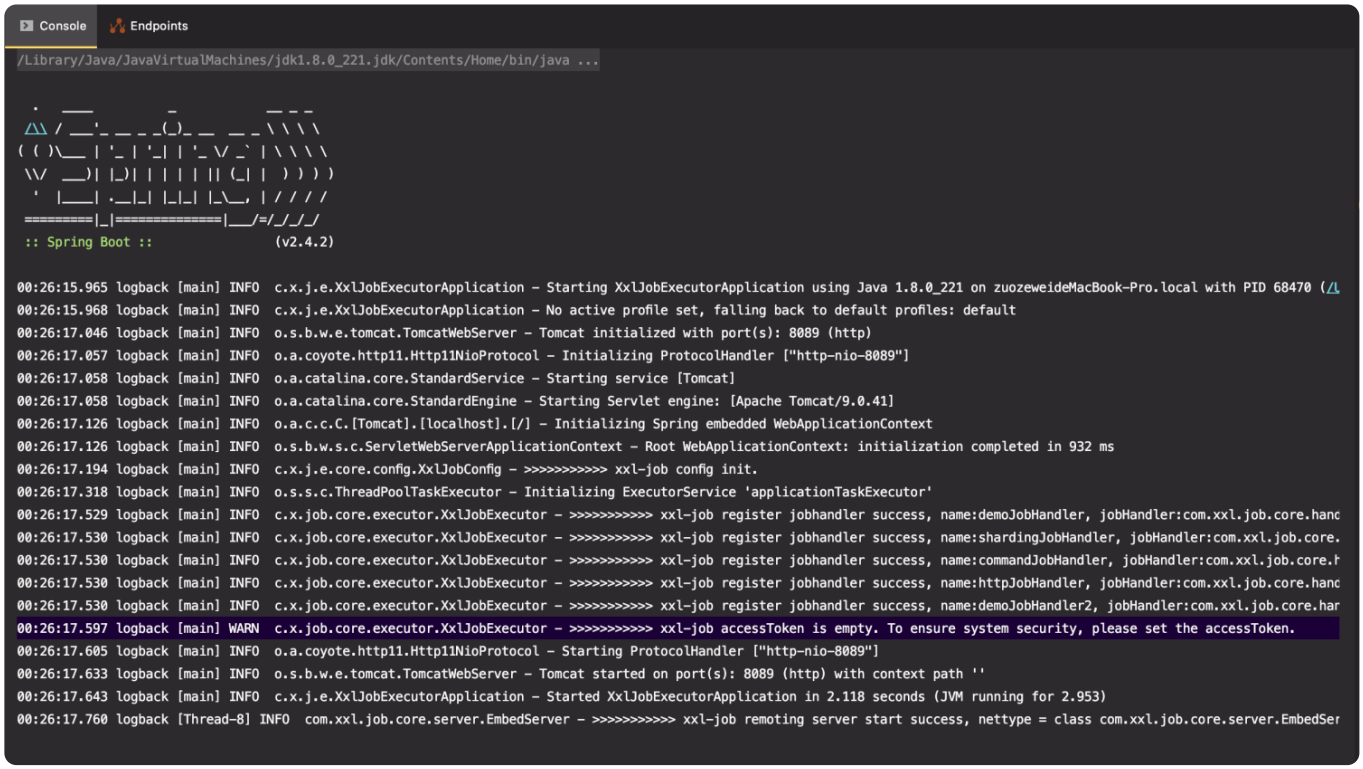


然后我们需要修改配置文件 application.properties：

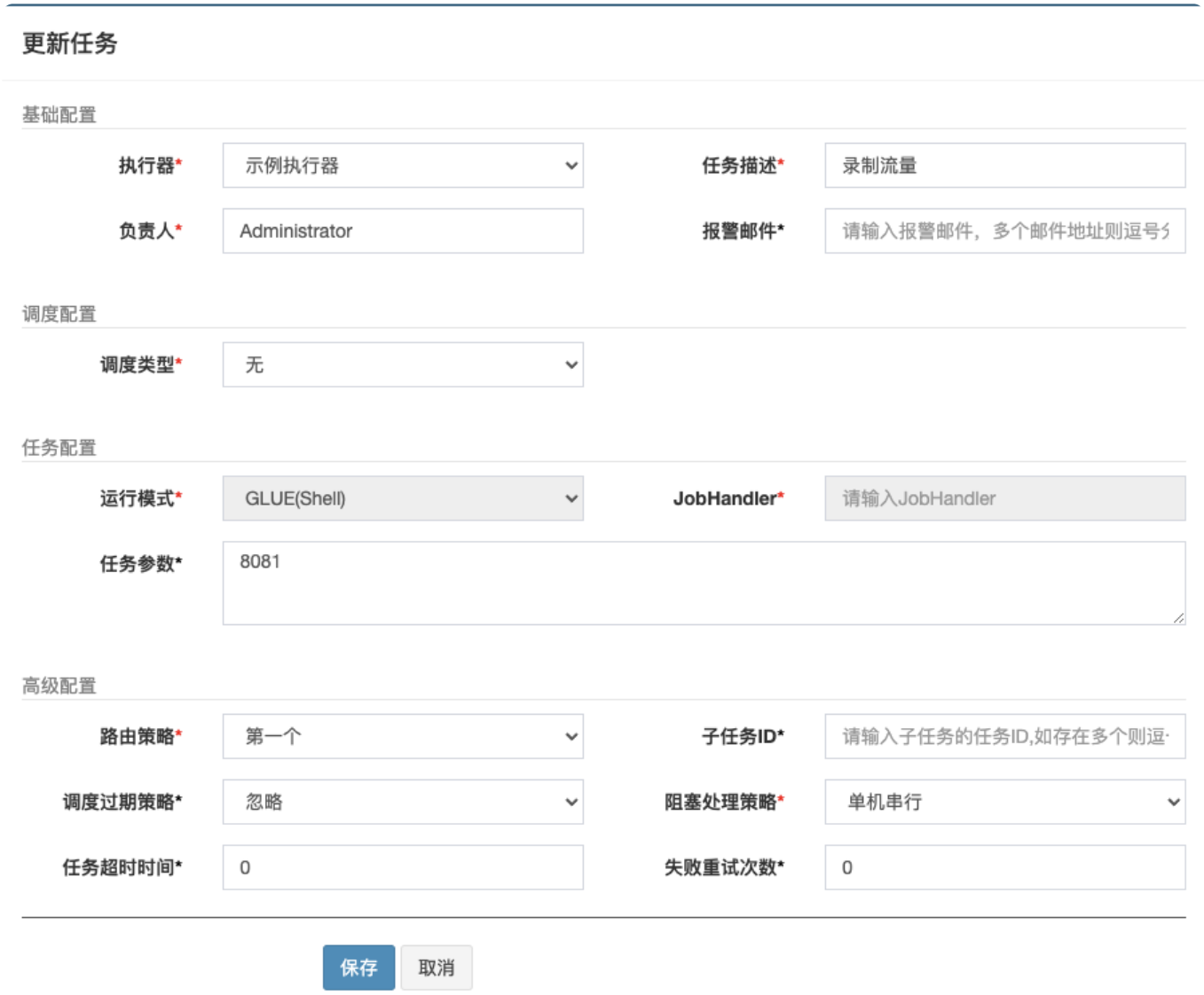
复制代码

```
# web port
server.port=8089
# no web
#spring.main.web-environment=false

# log config
logging.config=classpath:logback.xml

### 注册到调度中心
### xxl-job admin address list, such as "http://address" or "http://address01,
xxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin

### xxl-job, access token
xxl.job.accessToken=

### 集群部署，这两项配置要一致
### xxl-job executor appname
xxl.job.executor.appname=xxl-job-executor-sample
### xxl-job executor registry-address: default use address to registry , other
xxl.job.executor.address=

### xxl-job executor server-info
xxl.job.executor.ip=127.0.0.1
### 如果要模拟执行器集群部署，打包后单击运行多次，为服务设置随机端口，${random.int[10000,
xxl.job.executor.port=9999
### 日志存放路径
### xxl-job executor log-path
xxl.job.executor.logpath=./data/applogs/xxl-job/jobhandler
### xxl-job executor log-retention-days
xxl.job.executor.logretentiondays=30
```

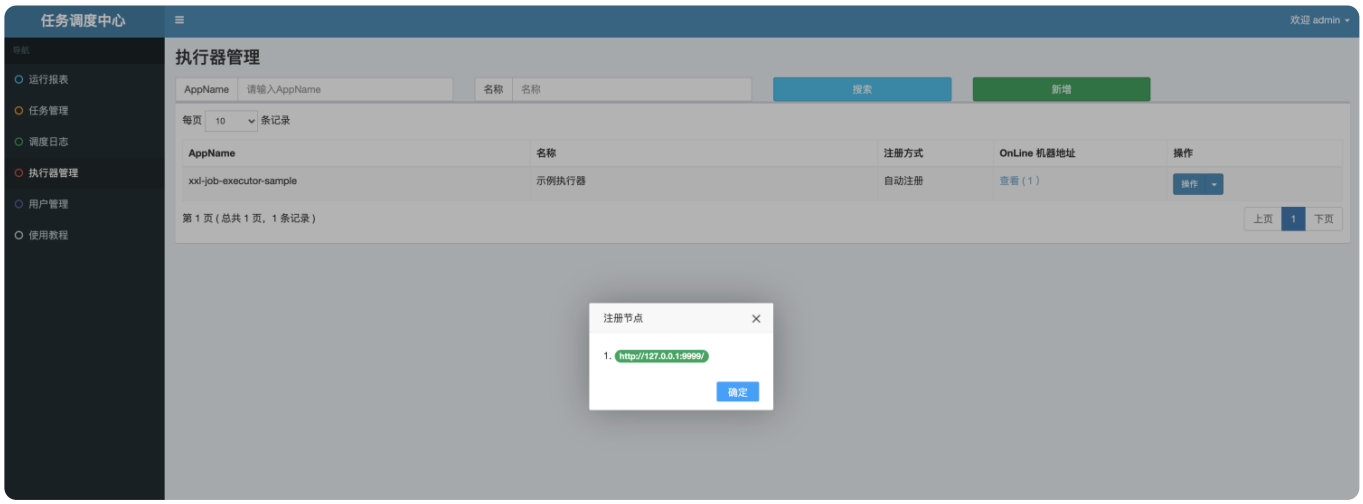
接着，我们在 IDEA 启动项目。



最后，我们在调度中心的【执行器管理】模块里新增一个执行器。



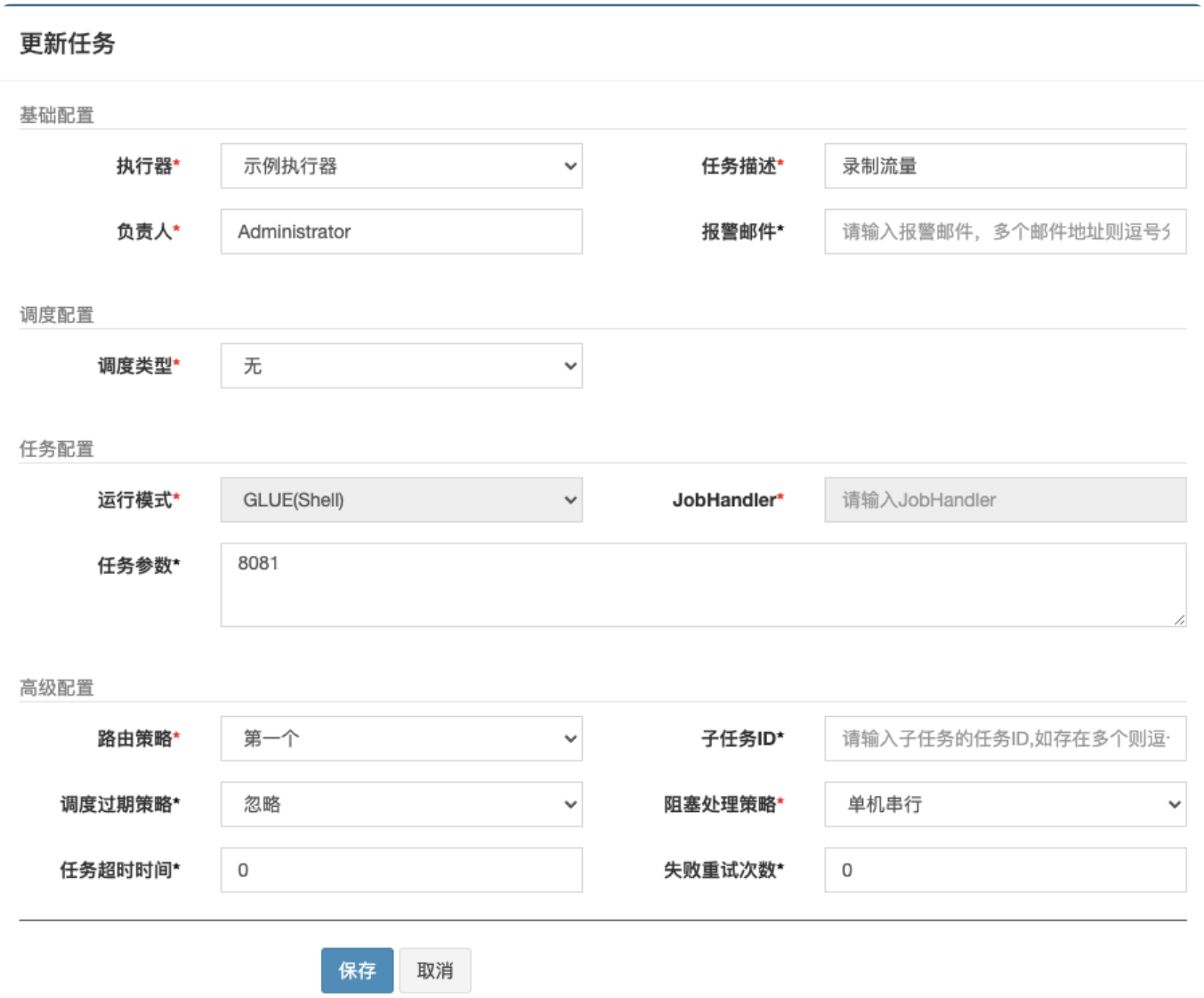
刷新之后就可以看到我们注册好的的执行器了。



到这里，我们的执行器就已经成功被调度中心注册和发现了，下面我们需要创建对应的任务。

创建流量录制任务

我们在调度中心新建一个调度任务。这里的运行模式我们选择 GLUE 模式 (Shell)。



简单说明一下，运行模式主要分为两种，一种是 BEAN，一种是 GLUE：

BEAN 模式：在项目中写 Java 类，然后在 JobHandler 里填上 @XxlJob 里面的名字，它是在执行器端代码编写的；

GLUE 模式：支持 Java、Shell、Python、PHP、Nodejs、PowerShell，以源码方式直接维护在调度中心，该任务实际上是一段对应脚本的执行。

我们知道 GoReplay 的所有操作都是通过命令行组织的，所以 GLUE 模式的 Shell 脚本是满足我们的需求的，且能做到集中式管理。

首先创建任务，点击该任务右侧的“GLUE”按钮，前往 GLUE 任务的 Web IDE 界面。这个界面支持对任务代码进行开发（也可以在 IDE 中开发完成后，复制粘贴到 Web IDE 中）。

你可以看看我们在 IDE 中开发的流量录制命令。

复制代码

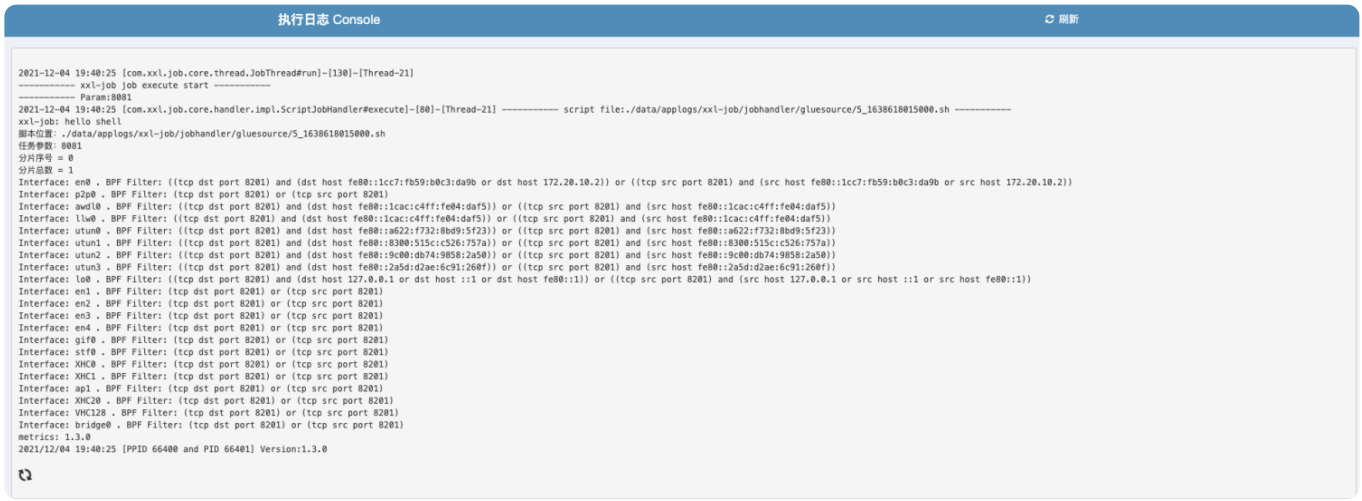
```
#!/bin/bash
echo "xxl-job: hello shell"

echo "脚本位置：$0"
echo "任务参数：$1"
echo "分片序号 = $2"
echo "分片总数 = $3"

cd /Users/GoProjects/src/goreplay/examples/middleware
./goreplay --input-raw :8201 --output-file=request-mall-all.gor -output-file-a

echo "Good bye!"
exit 0
```

刚才我们说过，因为 GLUE 这个模式的任务实际上是一段 “Shell” 脚本，所以任务建好后会存在一个安全问题：我们没有做鉴权。要解决这个问题方法也很简单，只需要在调度中心和执行器的 application.properties 里加上相同的 token 即可。

任务建好后，在执行时，我们可以传入任务参数，这里我们传入的是监听端口号 8081。

好了，开发完成后，我们点击启动：

点击执行日志右侧的 “执行日志” 按钮，可跳转至执行日志界面查看业务代码中打印的完整日志：

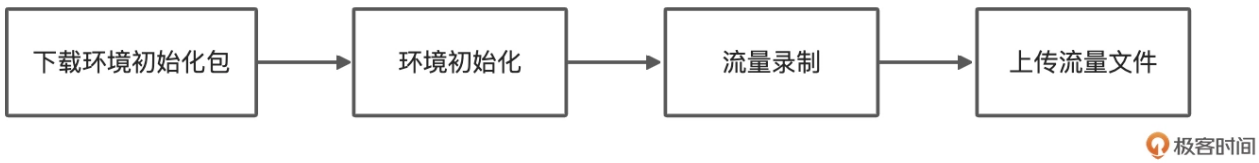


点击“日志”按钮，可以查看任务历史调度日志，每次任务调度的调度结果、执行结果等。点击“执行日志”按钮可查看执行器完整日志。

好了，到这里我们的流量录制任务就已经定义完了。至于流量回放、文件上传等其它任务的套路和这个类似。你可以按照差不多的步骤进行操作，遇到问题的话也可以来跟我讨论。

使用子任务

当我们需要做流量录制时，首先我们需要执行程序初始化任务，然后再执行流量录制任务，最后，我们需要将产生的流量文件上传到对象存储（流量仓库）中。那么，我们要怎么保证数据在三个任务之间顺利传递呢？

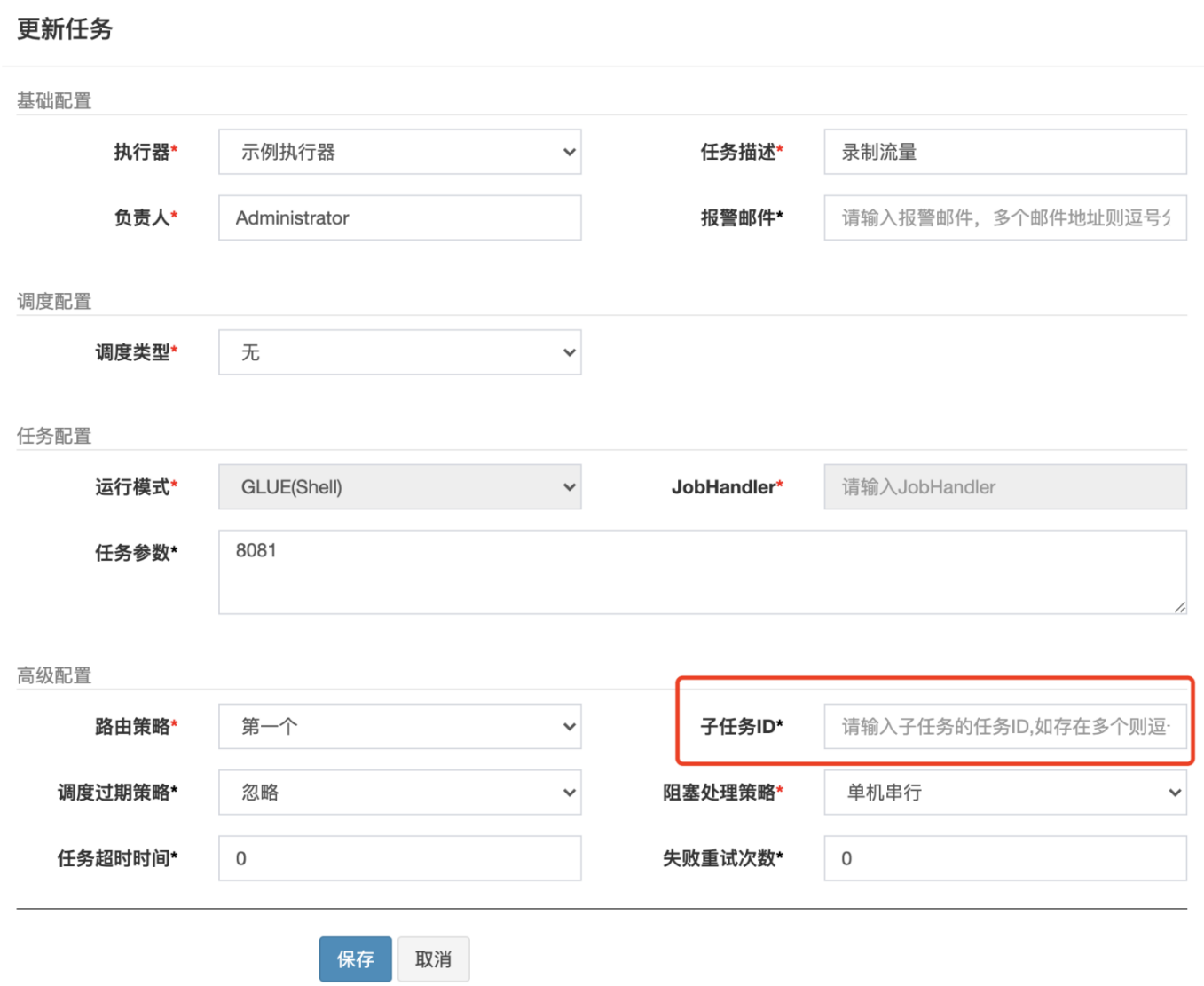


解决这种问题的思路一般有两种：

第一种就是写成一个大 job，让任务串行化执行；

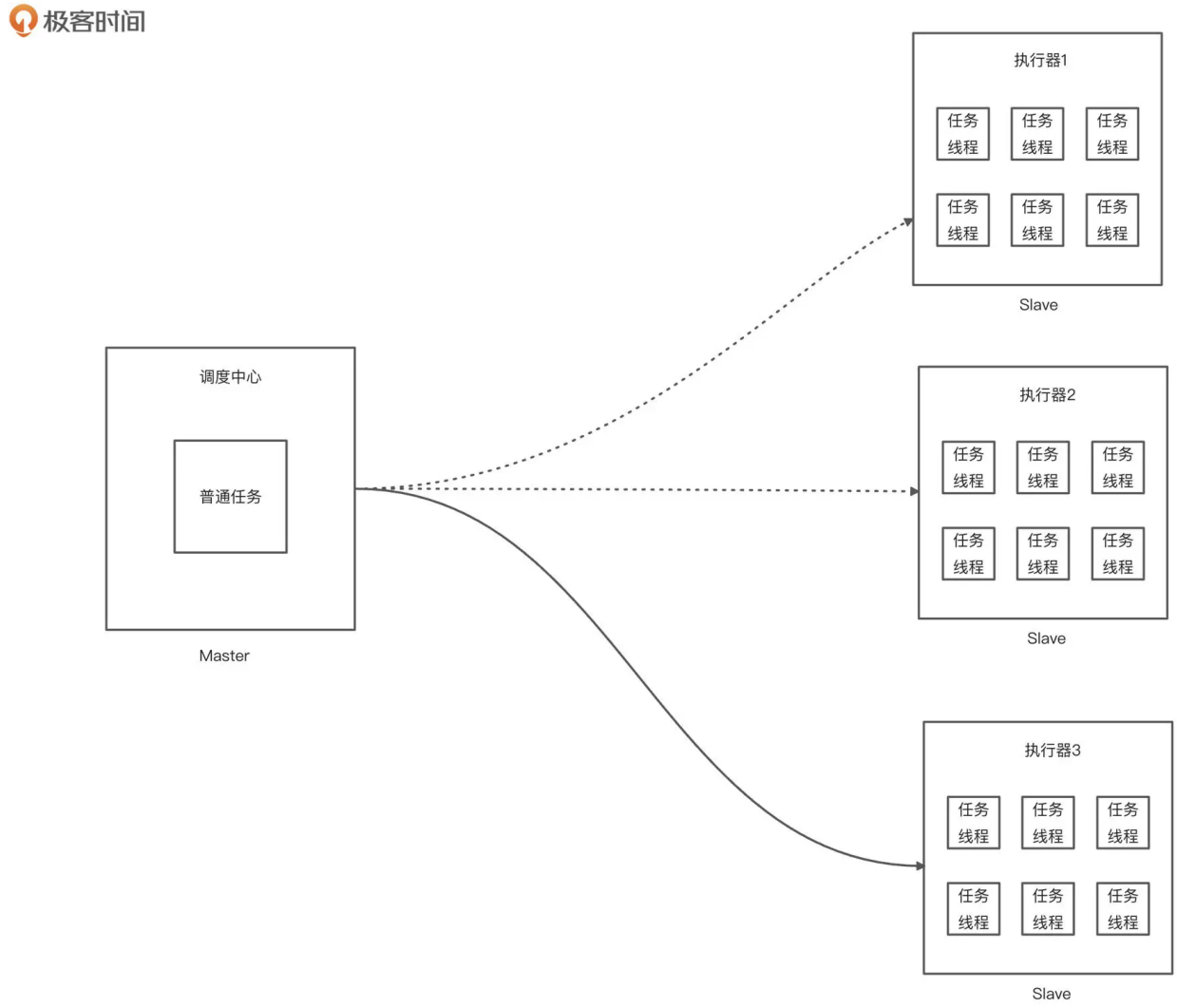
第二种就是使用多个子任务，在一个上游任务末尾触发另一个子任务。

如果我们想要在本任务成功执行后触发另外一个任务，可以把另外的任务作为本任务的子任务执行，它的操作也很简单，只需要在子任务一栏填上任务 id 即可：

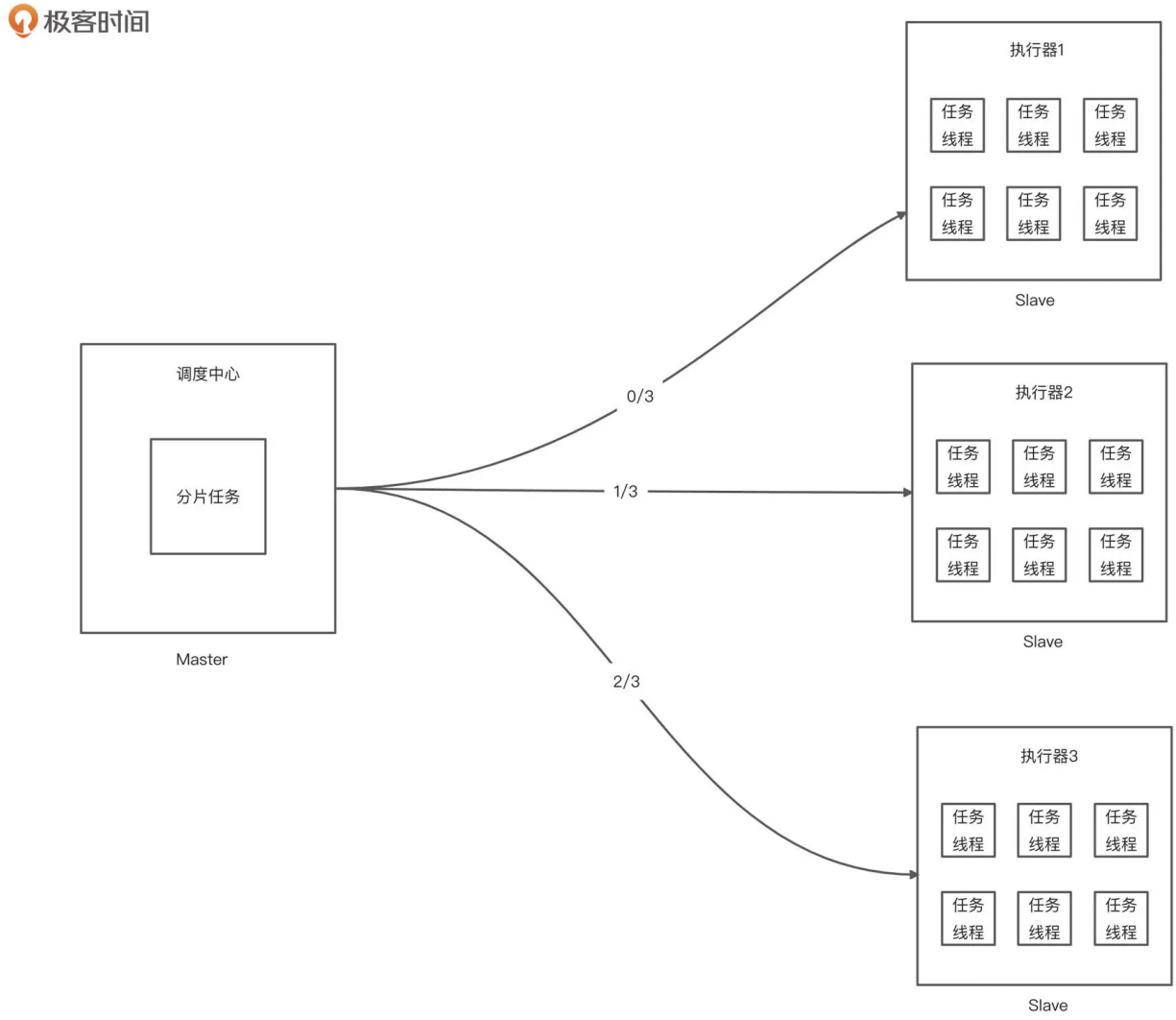


压测任务分片

全链路压测中一个显著的特点就是能够实现海量数据的并发请求，但这对于单线程的普通的任务来说是存在瓶颈的。这个时候如果你想提高并发，那么自然就要多线程跑嘛。但是用多线程跑数据也会存在问题，比如我开 3 个执行器，但你会发现只有个别执行器在拼命工作，其它的都处在休息状态。这是一个典型的任务分配不均的问题。



这时候就要用到我们的分片任务了，也就是要多台执行器共同完成任务。



但这里又有一个问题。比如流量回放的时候，多台压力机都执行同一个流量文件，相同的请求三台压力机都发一遍，那岂不是乱套了。

这里的解决思路也很简单，我们让每台执行器处理 1/3 的流量，大家把需要回放的流量文件平均切分了嘛，我干 1/3，你干 1/3，它干 1/3，这样就不会产生冲突了。

分片任务在运行的时候，调度器会给每个执行器发送一个不同的分片序号，分片的最大序号跟执行器的总数量是一样的，这样可以确保每个执行器都会执行到这个任务。

比如上图中第一个执行器拿到分片序号 0，第二台执行器拿到分片序号 1，第三台执行器拿到分片序号 2。那现在就好办了，我们只需要把处理的流量文件进行模 3 取余，余数为 0 的文件就由执行器 1 跑，余数为 1 的文件就由执行器 2 跑，余数为 2 的文件就由执行器 3 跑。

具体的解决方案，我们可以使用路由策略「分片广播」来调度任务：

1. 通过“任务参数”传入执行任务压力机数量；
2. 在定时任务逻辑中，根据获取到的分片参数、执行任务压力机数量，决策当前节点是否需要执行，切分流量文件并处理：
   - 如果 分片序号 > (执行压力机数量 - 1)，则当前节点不执行任务，直接返回；
   - 否则，取「分片序号」和 「压力机数量」 作为分片参数，流量文件并处理。

这样，我们可以实现灵活调度 [1, N] 个节点并行执行压测任务。

我们切分流量文件的代码可以这样写。

复制代码

```
package com.xxl.job.executor.until;

import cn.hutool.core.io.file.FileReader;
import cn.hutool.core.io.file.FileWriter;
import org.junit.jupiter.api.Test;
import org.junit.platform.commons.util.StringUtils;

import java.util.*;

/**
 * @author dunshan
 * @description: 分割流量文件
 * @date 2021-12-04 11:11:11
 */
public class NcountGor {



    @Test
    public void countTxt() {
        //分组
        int n = 3;

        LinkedHashSet<String> total = new LinkedHashSet<>();
        //读取文件
        FileReader fileReader = new FileReader("/Users/2021/10/request-mall-al
        String result = countTotal(total, fileReader);
        System.out.println("total:" + total.size());
        //放入 list 中进行分组使用
        List<String> list = new ArrayList<>();
        //组装分割文件
```

```
        for (String str : total) {
            list.add(str);
        }
        saveFile(n, result, list);

    }

    /**
     * 根据分割数据把文件切割文件
     *
     * @param n      切割数据
     * @param result 文件内容
     * @param list   分组数据
     */
    private void saveFile(int n, String result, List<String> list) {
        int number = (list.size() / n) + 1;
        List<List<String>> lists = groupList(list, number);
        for (int i = 0; i < lists.size(); i++) {
            //保存文件
            String path = String.format("/Users/2021/10/11/request_%s.gor", i)
            FileWriter writer = new FileWriter(path);
            for (int j = 0; j < lists.get(i).size(); j++) {
                //分割数据
                String[] content = result.split("\uD83D\uDC35\uD83D\uDE48\uD83
                for (String s : content) {
                    if (s.contains(lists.get(i).get(j))) {
                        writer.append(s);
                        //拼接切割文件
                        writer.append("\uD83D\uDC35\uD83D\uDE48\uD83D\uDE49");
                    }
                }
            }
        }
    }

    /**
     * 统计流量文件总数
     *
     * @param total
     * @param fileReader
     * @return
     */
    private String countTotal(LinkedHashSet<String> total, FileReader fileRead
        String result = fileReader.readString();
        String[] twoLine = result.split("\r\n");
        for (String str : twoLine) {
            if (StringUtils.isNotBlank(str)) {
                //转换字符数组
                char[] chars = str.toCharArray();
                //判断第一个字母是否是一
                if (String.valueOf(chars[0]).equals("1")) {
                    total.add(str.substring(2, 26));
```

```
                }
            }
        }
        return result;
    }

    /**
     * 流量文件分组
     *
     * @param list 流量文件
     * @param n    每个数组多少
     * @return 分组
     */
    private static List<List<String>> groupList(List<String> list, Integer n)
        // 求余数
        int remainder = list.size() % n;
        // 求分组数
        int count0 = list.size() / n;
        boolean flag = false;
        if (remainder >= n / 1) {
            flag = true;
        }
        List<List<String>> data = new ArrayList<>();
        // 获取多余 list
        List<String> sub = new ArrayList<>(list.subList(list.size() - remainde
        // 遍历 list 到余数前
        for (int i = 0; i < count0 + 1; i++) {
            List<String> ls;
            if (flag) {
                if (i == count0) {
                    break;
                }
                ls = new ArrayList<>(list.subList(i * n, i * n + n));
                // 将余出的数据加入分组
                if (i < sub.size()) {
                    ls.add(sub.get(i));
                }
            } else {
                int endex = i * n + n;
                if (endex >= list.size()) {
                    endex = list.size();
                }
                ls = new ArrayList<>(list.subList(i * n, endex));
            }
            if (ls.size() != 0) {
                data.add(ls);
            }
        }
        return data;
    }

}
```

切分的文件效果如下：

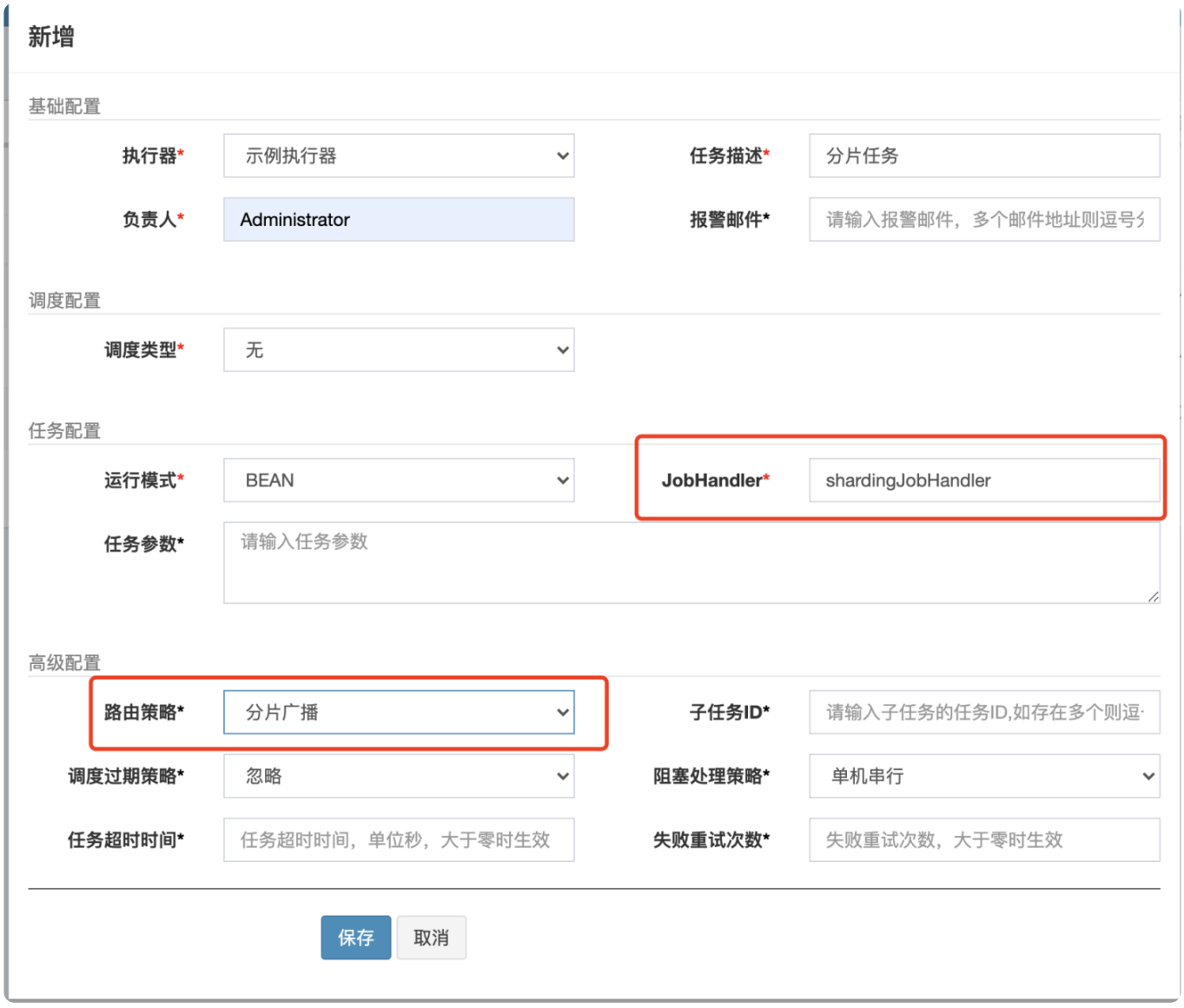
JobHandler 中的实现可以参考下面这种写法：

复制代码

```
/**
    * 2、分片广播任务
    */
   @XxlJob("shardingJobHandler")
   public void multiMachineMultiTasks() throws Exception {
       String param = XxlJobHelper.getJobParam();
       if (StringUtils.isBlank(param)) {
           XxlJobHelper.log("任务参数为空");
           XxlJobHelper.handleFail();
           return;
       }

       //读取文件
       FileReader fileReader = new FileReader(param);
       //总分片数
       String shardTotal = countTotal(total, fileReader);
       //分片参数
       int shardIndex = XxlJobHelper.getShardIndex();

       XxlJobHelper.log("分片参数：当前分片序号 = {}, 总分片数 = {}", shardIndex,

       LinkedHashSet<String> total = new LinkedHashSet<>();

       //放入 list 中进行分组使用
       List<String> list = new ArrayList<>();
       //组装分割文件
       for (String str : total) {
           list.add(str);
       }
       saveFile(n, result, list);

       // 业务逻辑
       for (int i = 0; i < shardTotal; i++) {
```

```
34              if (i == shardIndex) {
35                  XxlJobHelper.log("第 {} 片，命中分片开始处理", i);
36                  //分片流量回放
37                  process(shardIndex, shardTotal);
38              } else {
39                  XxlJobHelper.log("第 {} 片，忽略", i);
40              }
41          }
42
43      }
```

新建任务的时候，我们选择分片任务，填上对应的 JobHandler 即可。



最后需要说明一下，分片的数据量不一定是完全均等的，上面的取模只是一个例子、一个思路。我们也可以把 0、1、2 替换成其他条件去从所有数据中获取部分数据。比如分片序号是 0 的压力机回放网关 1 的流量文件，分片序号是 1 的压力机回放网关 2 的流量文件，分片序号是 2 的压力机我回放网关 3 的流量文件，具体怎么分我们可以自由选择。

通用 RESTful API 支持

我们知道，线上压测是一项对风险管控要求更高的活动，压测平台需要有更及时的异常预警机制，也就是说，必须要有压测实时监控和异常熔断能力。这就要求我们的调度任务能实时对接第三方的监控系统。

而 XXL-JOB 目标就是一种跨平台、跨语言的任务调度规范和协议。针对第三方监控系统，比如 Prometheus，我们可以借助 XXL-JOB 的标准 RESTful API 方便地实现任务回调支持。

标准 RESTful API 主要包括两部分：

- 调度中心 RESTful API：调度中心提供给执行器使用的 API；不局限于官方执行器使用，第三方可使用该 API 来实现执行器。主要包括：执行器注册、任务结果回调等 API；
- 执行器 RESTful API：执行器提供给调度中心使用的 API；官方执行器默认已实现，第三方执行器需要实现并对接提供给调度中心，任务触发、任务终止、任务日志查询等 API。

除此之外，如果我们有需要通过 API 操作调度中心，可以个性化扩展 “调度中心 RESTful API” 并使用。

总结

流量平台是一个全链路压测项目的引擎。但对于一个大企业来说，分布式的流量平台是一个必要的条件。

在全链路分布式流量平台改造中，我强调了几个特性：高可靠、高性能、低成本、高效率。

我们在做技术改造的时候，一定要牢记下面几项关键原则：

- 基于开源，拥抱开源，尽量使用成熟的第三方框架，不重复造轮子；
- 技术栈和组件要适合研发团队技术能力及主流技术方向；

KISS 原则 (Keep It Simple and Stupid)；

尽量考虑 ROI（投入产出比）；

不断演进迎合公司业务发展。大公司方案不一定是适合的，要避免为技术而技术；

任何方案不必追求大而全，可以在实践中不断完善和演进；

需求导向，借力开源，科学选型，快速集成，重视扩展，演进改善。

刚才，我还把一个分布式流量平台的需求以及需要改造到什么地步，都给你梳理了一遍。希望能给你一些借鉴。

思考题

学完这节课，请你思考两个问题：

1. 你有没有接触过分布式流量平台的改造工作，请谈谈你的心得。
2. 关于分布式调度框架，你有没有接触过其它的项目？

欢迎你在留言区与我交流讨论。当然了，你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 压测平台：如何改造对象存储和性能监控？

下一篇 25 | 环境搭建：我们的系统是怎么搭建起来的？

更多学习推荐

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。